CS 314 Principles of Programming Languages

Fall 2017

A Compiler and Optimizer for tinyL

Due date: Monday, October 23, 11:59pm

THIS IS NOT A GROUP PROJECT! You may talk about the project and possible solutions in general terms, but must not share code. In this project, you will be asked to write a recursive descent LL(1) parser and code generator for the tinyL language as discussed in class. Your compiler will generate RISC machine instructions called ILOC (Intermediate Language for Optimizing Compilers). You will also write a code optimizer that takes ILOC instructions as input and implements different peephole optimizations. The output of the optimzer is a sequence of ILOC instructions which produces the same results as the original input sequence. To test your generated programs, you can use a virtual machine (simulator) that can "run" your ILOC programs. The project will require you to manipulate linked lists of instructions. In order to avoid memory leaks, explicit deallocation of "eliminated" instructions is necessary.

This document is not a complete specification of the project. You will encounter important design and implementation issues that need to be addressed in your project solution. **Identifying these issues is part of the project**. As a result, you need to start early, allowing time for possible revisions of your solution.

<program>	::=	<stmt_list> .
<stmt_list>	::=	<stmt> <morestmts>
<morestmts>	::=	; <stmt_list> | ϵ
<stmt>	::=	<assign> | <print>
<assign>	::=	<variable> = <expr>
<print>	::=	! <variable>
<expr>	::=	+ <expr> <expr> |
		− <expr> <expr> |
		∗ <expr> <expr> |
		/ <expr> <expr> |
		<variable> |
		<digit>
<variable>	::=	a | b | c | d | e | f | g | h | i | j | k | x | y | z
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 1: The tinyL language as specified by a context-free grammar

1 Background

1.1 The tinyL language

tinyL is a simple expression language that allows assignments, and print as its only I/O operation. Every token is a **single** character of the input. This makes scanning rather easy, but does not allow integer constants of more than one digit, or variable names of more than one character. The language specification in Backus-Naur from is given in Figure . The following are examples of two valid **tinyL** programs:

(1) `a=3;b=5;c=/3*ab;d=+c1;!d.`
(2) `a=7;b=-*+1+2a58;!b.`

1.2 Target Architecture

instr. format	description	semantics
memory instructions		
`loadI` $c \Rightarrow r_x$	load constant value $c$into register r_x	$r_x \leftarrow c$
`loadAI` r_x, $c \Rightarrow r_y$	load value of MEM($r_x + c$) into r_y	$r_y \leftarrow$ MEM($r_x + c$)
`storeAI` $r_x \Rightarrow r_y$, c	store value in $r_x$into MEM($r_y + c$)	MEM($r_y + c$) $\leftarrow r_x$
bit-shift operations		
`lshiftI` r_x, $c \Rightarrow r_z$	shift contents of registers $r_x$by c positions to the left store result into register r_z	$r_z \leftarrow r_x << c$
`rshiftI` r_x, $c \Rightarrow r_z$	shift contents of registers $r_x$by c positions to the right store result into register r_z	$r_z \leftarrow r_x >> c$
arithmetic instructions		
`add` r_x, $r_y \Rightarrow r_z$	add contents of registers $r_x$and r_y, and store result into register r_z	$r_z \leftarrow r_x + r_y$
`sub` r_x, $r_y \Rightarrow r_z$	subtract contents of register $r_y$from register r_x, and store result into register r_z	$r_z \leftarrow r_x - r_y$
`mult` r_x, $r_y \Rightarrow r_z$	multiply contents of registers $r_x$and r_y, and store result into register r_z	$r_z \leftarrow r_x * r_y$
`div` r_x, $r_y \Rightarrow r_z$	divide contents of registers $r_x$and r_y, and store result into register r_z	$r_z \leftarrow r_x / r_y$
I/O instruction		
`outputAI` r_x, c	write value of MEM($r_x + c$) to standard output	print(MEM($r_x + c$)

The target architecture is a RISC machine with 4096 registers. All registers can only store integer values. A RISC architecture is a load/store architecture where arithmetic instructions operate on registers rather than memory operands (memory addresses). This means that for each access to a memory location, a `load` or `store` instruction has to be generated. Here is the machine instruction set of our RISC target architecture. You are only allowed to use these ILOC instructions. ILOC instructions are case sensitive. r_x, r_y, and $r_z$represent three registers.

1.3 Code Shape

Your compiler should generate code of a specific form with respect to how variables are accessed. All variables accesses use an address that consists of a **base pointer** and an **offset** relative to this base pointer. The base pointer address is stored in a special register, in our case r_0. All memory references are therefore of the form MEM(r_0 + offset). This is what the instructions `loadI, loadAI, storeAI` and `outputAI` use. All addresses are **byte** addresses. Your compiler should assign the address 1024 to r_0 at the beginning of the program. For our example language, variable offsets are non-negative byte addresses. Your compiler should map variable "a" to offset 0, variable "b" to offset 4, variable "c" to offset 8, etc. Other mappings are also possible, so we are really talking about "code shape" here, which is a particular coding style.

Your compiler should generate code that does not "reuse" registers. If you assign a value to a register by a `loadI, loadAI, add, sub, mult` or `div` instruction, you will always use a fresh, i.e., new register. Your target machine has many registers, so do not worry about running out of registers. For example, this means that the code generated for `+ 1 1` will generate two `loadI` instructions with two distinct target registers and one `add` instruction. This coding style is also called the register-register model, where each computed value gets its own register. In a real compiler, an additional optimization pass maps (virtual) registers to the limited number of physical registers of a machine. This step is typically called *register allocation*. We do not deal with register allocation here.

Our tinyL language does not contain any control flow constructs (e.g.: jumps, if-then-else, while). This means that every generated instruction in the instruction sequence will be executed.

2 Project Description

The project consists of two main parts:

1. Complete the partially implemented recursive descent LL(1) parser that generates ILOC instructions.

2. Write a peephole optimizer for constant folding and operator strength reduction.

In addition, you are asked to write the `PrintInstructionList` routine. The project represents an entire programming environment consisting of a compiler, an optimizer, and a simulator (virtual machine) for ILOC. The ILOC simulator is called **sim** and will be made available to you as an executable on the ilab machines. This will allow you to check for correctness of your generated and optimized code.

2.1 Compiler

The recursive descent LL(1) parser implements a simple code generator. You should follow the main structure of the code as given to you in file `Compiler.c`. As given to you, the file contains code for function `digit`, `variable`, and partial code for function `expr`. As is,

the compiler is able to generate code only for expressions that contain "+" operations on operands that are digits or the variable "f". You will need to add code in the provided stubs to generate correct RISC machine code for the entire program. Do not change the signatures of the recursive functions. Note: The left-hand and right-hand occurrences of variables are treated differently.

2.2 I/O Instruction Utility

Within the **Optimizer**, a sequence of ILOC instructions is represented as a doubly-linked list. You are asked to implement the following utility function in file `InstrUtils.c`.

```
void PrintInstructionList(FILE *outfile, Instruction *instr);
```

Function `PrintInstructionList` traverses the instruction list beginning with instruction "instr". The list is written into file "outfile". The implementation of this function **must be based on** the utility function

```
void PrintInstruction(FILE *outfile, Instruction *instr);
```

The implementation of the latter function is provided to you in file `InstrUtils.c`. This is also the file that will contain your implementation of `PrintInstructionList`

2.3 Peephole Optimization

There are two types of peephole optimizations that you will need to implement: constant folding and operator strength reduction. Peephole optimizations look for particular instruction subsequences that can be replaced by a more efficient instruction sequence.

The peephole optimizer uses a sliding window of three RISC machine instructions. It looks for code patterns as described below. If no pattern is detected, the window is moved one instruction down the list of instructions. In the case of a successful match and code replacement, the first instruction of the new window is set to the instruction that immediately follows the instructions that have been replaced, i.e., is set to the instruction immediately after the pattern in the unoptimized code.

A peephole optimization pass (constfolding or strengthreduct) expects the ILOC input file to be provided at the standard input (stdin), and will write the generated code back to standard output (stdout). This allows the specification of multiple passes of the same optimization or different sequences of optimization passes using the UNIX pipe feature. For example, to apply optimization constfolding twice, followed by a strengthreduction pass strengthreduct with file "tinyL.out" as input, and file "optimized.out" as output, we would specify

```
./constfolding < tinyL.out | ./constfolding | ./strengthreduct > optimized.out
```

Instructions that are deleted as part of the optimization process have to be explicitly deallocated using the C `free` command in order to avoid memory leaks. You will implement your two peephole optimization passes in file `ConstFolding.c` and `StrengthReduction.c`.

2.3.1 Constant folding

Patterns are consecutive instructions in your ILOC program. The pattern for constant folding is a follows:

```
loadI c1 ⇒ ra
loadI c2 ⇒ rb
OP ra, rb ⇒ rc
```

where c_1 and c_2 are integer constant, and `OP` is an arithmetic operation (`add, sub, mult`). Note: We do not apply this optimization for the division operator. The above pattern can be replaced by the code sequence

```
loadI c3 ⇒ rc
```

where c_3 is the integer constant that represents the result of the computation (c_1 OP c_2), which is done at compile time. Due to the code shape assumption, register values in r_a and r_b are not used after the optimized pattern, allowing the two `loadI` instructions to be deleted.

2.3.2 Operator strength reduction

The patterns for this optimization are as follows:

```
loadI c1 ⇒ ra
OP rb, ra ⇒ rc
```

where c_1 is an integer constant and a power of 2 (e.g.: 2, 4, 8, 16, ...) and `OP` is the arithmetic operation `mult` or `div`. Multiplication and division can be implemented as left-shift or right-shift operations, respectively. For example,

```
loadI 4 ⇒ ra
div rb, ra ⇒ rc
```

can be replaced by

```
rshiftI rb, 2 ⇒ rc
```

and

```
loadI 4 ⇒ ra
mult rb, ra  ⇒ rc
```

can be replaced by

```
lshiftI rb, 2 ⇒ rc
```

Note: Arithmetic integer operations may result in overflow or underflow conditions due to the finite bit-width of the integer representation. However, this is not a problem for these peephole optimizations since the overflow/underflow would also occur in the unoptimized code.

2.4 ILOC Simulator

The virtual machine executes ILOC program. If a `outputAI <id>` instruction is executed, the value of the specified memory location is written to standard output (stdout). All values are of type integer. An ILOC simulator is provided as an executable (**sim**). The ILOC simulator reports the overall number of executed instructions for a given input program. This allows you to assess the effectiveness of your dead code elimination optimization. You also will be able to check for correctness of your optimization pass.

3 Grading

You will submit your versions of files `ConstFolding.c`, `StrengthReduction.c`, `Compiler.c` and `InstrUtils.c`. You may also submit an optional `ReadMe` file if you want to communicate something about your code to the grader. **No other file should be modified, and no additional file(s) may be used**. The electronic submission procedure will be posted later. **Do not submit any executables or any of your test cases**.

Your programs will be graded based mainly on functionality. Functionality will be verified through automatic testing on a set of syntactically correct test cases. **No error handing is required**. The original project distribution contains some test cases. Note that during grading we will use additional test cases not known to you in advance. The distribution also contains executables of reference solutions for the compiler (compile.sol) and the optimizers (constfolding.sol, strenghtreduct.sol), and the iloc simulator sim. A simple Makefile is also provided in the distribution for your convenience. For example, in order to create the compiler, say `make compile` at the Linux prompt, which will generate the executable compile.

The provided, initial compiler is able to parse and generate code for very simple programs consisting of a single assignment statement with right-hand side expresssions of only additions of numbers, followed by a single print statement. You will need to be able to accept and compile the full tinyL language.

The Makefile also contains rules to create executables of your optimization passes.

4 How To Get Started

The code for this project lives on the ilab cluster in directory:

`www.cs.rutgers.edu/courses/314/classes/fall_2017_kremer/projects/proj1/students`

Create your own directory on the ilab cluster, and copy the entire provided project `proj1.tar` to your own home directory or any other one of your directories. Say `tar -xf proj1.tar` to extract the project files. Make sure that the read, write, and execute permissions for groups and others are disabled (`chmod go-rwx <directory_name>`). **IT IS CONSIDERED CHEATING IF YOU DO NOT PROTECT YOUR PROJECT FILES**.

Say `make compile` to generate the compiler. To run the compiler on a test case "test", say `./compile test`. This will generate a RISC machine program in file `tinyL.out`. To create an optimization pass, say `make constfolding` or `make strengthreduct`. The distributed versions of the two optimization passes do not work at all, and the compiler can only handle a single example program structure consisting of a single assignment statement followed by a print statemet. An example test case that the provided compiler can handle is given in file `tests/test-dummy`.

To call your optimization pass `constfolding` on a file that contains RISC machine code, for instance file `tinyL.out`, say `./constfolding < tinyL.out > optimized.out`. This will generate a new file `optimized.out` containing the output of your optimizer. The operators "<" and ">" are Linux redirection operators for standard input (stdin) and standard output (stdout), respectively. The same holds for your `strenthreduct` optimization pass.

You may want to use valgrind for memory leak detection. We recommend to use the following flags, in this case to test the optimizer for memory leaks:

```
valgrind --leak-check=full --show-reachable=yes --track-origins=yes ./constfolding
< tinyL.out
```

To run a program on the ILOC simulator, for instance `tinyL.out`, say `./sim < tinyL.out`. Finally, you can define a **tinyL language interpreter** on a single Linux command line as follows:

```
./compile test; ./strengthreduct < tinyL.out > opt.out; ./sim < opt.out
```

The ";" operator allows you to specify a sequence of Linux commands on a single command line.

5 Questions

All questions regarding this project should be posted on piazza (sakai). Enjoy the project!